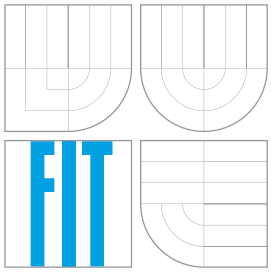
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA GRAMATIKÁCH ŘÍZENÝCH STROMY

PARSING BASED ON TREE-CONTROLED GRAMMARS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
ŠTĚPÁN GRANÁT

VEDOUCÍ PRÁCE
SUPERVISOR
Prof. RNDr. ALEXANDER MEDUNA, CSc.

BRNO 2016

Abstrakt

Cílem této práce je navrhnout a implementovat syntaktický analyzátor gramatik, jejichž derivační strom je omezen pomocí kontroly úrovní. Běžné postupy syntaktické analýzy jsou podrobně rozebrány a poté je diskutováno, jak by mohly být rozšířeny o kontrolu derivačního stromu. Nejdůležitější částí práce je návrh průběžné kontroly derivačního stromu souběžně s jeho konstrukcí, což umožňuje úzké propojení těchto dvou procesů. Uvedený přístup přináší výrazné zvýšení síly syntaktického analyzátoru.

Abstract

The goal of this thesis is to design and implement the parser of grammars, whose derivation tree is limited by inspection of levels. Common parsing procedures are studied in detail and then it is discussed, how they could be extended by inspection of derivation tree. The most important part of the thesis is a draft of continuous inspection of the derivation tree simultaneously with its construction, which allows close cooperation between these two processes. This approach enables significant increasing of the parser power.

Klíčová slova

gramatiky řízené stromy, syntaktická analýza, bezkontextové gramatiky

Keywords

tree controlled grammars, syntactic analysis, context-free grammars

Citace

GRANÁT, Štěpán. *Syntaktická analýza založená na gramatikách řízených stromy.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Syntaktická analýza založená na gramatikách řízených stromy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením profesora RNDr. Alexandera Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Štěpán Granát
12. května 2016

Poděkování

Děkuji prof. Alexanderu Medunovi za ochotu vést tuto práci, jeho odborné rady a morální podporu.

Obsah

Kapitola 1

Úvod

Teorie formálních jazyků si dává za cíl formalizovat jazyky a pojmy s nimi spojené tak, abychom mohli přesně definovat jejich pravidla, a tedy jednoznačně stanovit, jaké gramatické konstrukce jazyk umožňuje. Dalším krokem je vyvinout nástroje, které dokáží definici jazyka zpracovat a poté nad zadaným řetězcem rozhodnout, jestli do jazyka patří, či nikoliv. Takové nástroje se nazývají *syntaktické analyzátory* nebo *parsery*.

Syntaktický analyzátor má za úkol najít mezi jednotlivými symboly v řetězci vazby, které vychází z gramatických pravidel jazyka. Tyto vazby lze znázornit jako strom, jenž poté může být základem *sémantické analýzy*, tedy jakéhosi „pochopení“ přijímaného řetězce. Teorie formálních jazyků je základním kamenem při návrhu překladačů jazyků programovacích. Ty většinou spadají do *bezkontextových jazyků*, pro které již existují zavedené syntaktické analyzátory.

Někdy však síla těchto zavedených parserů nestačí, a proto se přistupuje k různým úpravám a vylepšením. Tyto úpravy mohou být buď jednoúčelové, nebo univerzálnějšího typu, kdy se snažíme celkově zvýšit vyjadřovací sílu syntaktického analyzátoru, při zachování determinismu a také co nejnižší asymptotické složitosti zpracování řetězce.

Tato práce pojednává o zpracování *gramatik řízených stromy*. Takové gramatiky se skládají ze dvou jazyků - řízeného a kontrolního. Řízený jazyk budeme definovat pomocí *bezkontextové gramatiky* a slouží k sestavení derivačního stromu pomocí již zavedených postupů. Kontrolní jazyk definujeme *konečným automatem* a poslouží k omezení derivačního stromu. Cílem je zvýšit vyjadřovací sílu syntaktického analyzátoru tak, abychom byli schopni zpracovávat co nejvíce jazyků, které nelze definovat (deterministickými) bezkontextovými gramatikami.

Pro realizaci parseru tohoto typu bylo nutné podrobně nastudovat problematiku *regulárních* a *bezkontextových jazyků*, čímž se zabýváme v kapitolách 2 a 3. Tyto teoretické poznatky jsou nezbytné pro pochopení účelu a výhod syntaktického analyzátoru založeného na gramatikách řízených stromy. Uvedené poznatky vychází převážně z knihy prof. Alexandra Meduny [5].

Hlavní částí práce je kapitola 4, kde je uvedena metoda kontroly úrovní derivačního stromu pomocí regulárního jazyka, jež je založena na práci Ing. Jiřího Koutného [4], který ve své práci zpracovává teoretické aspekty této problematiky.

Velmi detailně zde rozebíráme možnosti modifikace stávajících metod (LL a LR). Zkoumáme zde jejich výhody a nevýhody s ohledem na rozšíření o kontrolu úrovní, tak aby se činnost syntaktického analyzátoru a kontroly úrovní co nejvíce doplňovala. Na těchto poznatcích je založena experimentální metoda průběžné kontroly stromu, kdy propojujeme proces konstrukce stromu a jeho kontroly. Tento způsob umožňuje rychlejší odhalení chyb v derivačním stromě a hlavně si tyto metody jsou schopny navzájem pomáhat tak, že u nedeterministické řízené gramatiky kontrola úrovní pomůže při výběru pravidla. Tato technika je vysvětlena v sekci 4.3.

V poslední kapitole je stručně popsána implementace syntaktického analyzátoru, který využívá poznatky získané v předchozích kapitolách a na kterém byly prováděny pokusy s gramatikami uváděnými v této práci.

Kapitola 2

Teorie grafů

Tato práce předpokládá, že je čtenář seznámen s *Teorií grafů* a se základy *Teorie formálních jazyků* (viz [5]). Uvedeny jsou jen definice, jež jsou zásadní pro pozdější výklad.

Definice 2.0.1. (Strom)
Strom je orientovaný acyklický graf, $G = (\Sigma, R)$, který má tyto tři vlastnosti:

- G má právě jeden uzel, do něhož nevstupují žádné hrany; tento uzel se nazývá kořen G označovaný jako kořen(G).
- Jestliže $a \in \Sigma$ a $a \neq$ kořen(G), potom a je potomkem kořenu(G) a vstupuje do něj právě jedna hrana.
- Každý uzel $a \in \Sigma$, který není listem, má své přímé potomky, b_1 až b_n, řazené zleva doprava tak, že b_1 je nejlevějším přímým potomkem a a b_n je nejpravějším přímým potomkem a.

▲

Definice 2.0.2. (Úroveň, cesta, hranice, hloubka, elementární strom a podstrom)
Nechť $G = (\Sigma, R)$ je stromem.

- *Úroveň* l stromu G je posloupnost s všech uzlů se stejnou vzdáleností od kořene(G). Jinými slovy, úroveň l je posloupnost $s = n_1 n_2 ... n_k$ taková, že existuje cesta v grafu o délce ℓ v G pro všechny posloupnosti od kořene(G) $n_1 n_2 ... n_i$, pro $1 \leq i \leq k$ a $l \geq 1$.
- *Cesta* p stromu G je posloupnost s uzlů, kde první uzel je kořen(G), poslední je listem a mezi každými dvěma následnými uzly v s existuje hrana v G. Jinými slovy, cesta p stromu G je posloupnost $s = n_1 n_2 ... n_k$ taková, že s je cesta grafem o délce k v G, kde $n_1 =$ kořen(G) a n_k je listem v G, pro $k \geq 1$.
- *Hranice* stromu G, hranice(G), je posloupnost listů G řazených zleva doprava.
- *Hloubka* stromu G, hloubka(G), je délka nejdelší cesty v G; jestliže platí hloubka(G)= 1, potom je G *elementární strom.*
- Jestliže $G' = (\Sigma', R')$ představuje strom vyhovující těmto čtyřem podmínkám: $\Sigma' \neq \emptyset$; $\Sigma' \subseteq \Sigma$; $R' = (\Sigma' \times \Sigma')$; a jestliže v G není žádný z uzlů v $\Sigma - \Sigma'$ potomkem uzlu v Σ', potom je G' *podstromem* G.

▲

Kapitola 3

Formální jazyky

Teorie formálních jazyků formalizuje pojmy spojené s jazyky (přirozenými, programovacími, matematickými, ...), abychom se mohli zabývat jejich automatizovaným zpracováním. Pojmy, které známe z lingvistiky, jsou zde zobecněny a přesně definovány, takže nemusí úplně odpovídat tomu, jak jsou chápány v jiných vědních oborech.

Abeceda

Základem jazyka je *abeceda*. V teorii formálních jazyků je obvykle značena Σ (sigma) a je definována jako konečná neprázdná množina, jejíž objekty se nazývají *symboly*.

Řetězec

Konečná posloupnost symbolů patřících do Σ je *řetězec* nad Σ. Zvláštním případem je ε (epsilon), značící *prázdný řetězec* - tedy takový, že neobsahuje žádný symbol.

Jazyk

Σ^* značí množinu všech řetězců, které je možné sestrojit nad abecedou Σ. Jakákoliv podmnožina $L \subseteq \Sigma^*$ je *jazykem* nad abecedou Σ. Jestliže *jazyk* představuje konečnou množinu řetězců, potom jej nazýváme *konečným jazykem*, v opačném případě *jazykem nekonečným*.

Gramatika

Pokud se zabýváme nekonečnými jazyky, nemůžeme je vyjádřit jednoduchým výčtem jejich řetězců. Místo toho definujeme *gramatiku*, která stanovuje pravidla pro generování řetězců patřících do daného jazyka.

Gramatika obsahuje 4 části:

- Množinu *neterminálních symbolů* N (*neterminálů*), které slouží k označení syntaktických celků.
- Množinu *terminálních symbolů* Σ (*terminálů*) - symboly, které jsou konečným výstupem. (abeceda)
- Množinu *přepisovacích pravidel* P.
- Počáteční (startovací) symbol $S \in N$.

Přepisovací pravidla

Přepisovací pravidlo je složeno ze dvou řetězců (α, β), složených z terminálů a neterminálů, přičemž α obsahuje alespoň jeden *neterminál*. Zapisují se jako $\alpha \to \beta$.
Pravidla se aplikují od počátečního symbolu, kdy postupně přepisujeme řetězec tak, že nahradíme jakoukoliv část řetězce, která se nachází na levé straně některého pravidla, za pravou stranu tohoto pravidla. Tato operace se také nazývá *derivace*. Řetězec upravujeme podle pravidel tak dlouho, až se v něm nacházejí pouze *terminály*.

řetězec	pravidlo	výsledek
a**Q**e	Q -> bcd	a**bcd**e

Obrázek 3.1: Příklad derivace podle pravidla

Ekvivalence gramatik

Dvě gramatiky označujeme jako *ekvivalentní*, pokud generují stejný jazyk.

Výpočetní model

Výpočetní model lze definovat jako hypotetický přístroj, který obsahuje množinu povolených operací. Povolené operace modelu určují, jak sofistikované problémy je schopen řešit.

Hierarchie jazyků

Omezením gramatiky lze zaručit to, že ji lze zpracovávat jednodušším *výpočetním modelem*. Jazyky se proto dělí do tříd právě podle toho, jaký výpočetní model je dokáže zpracovat.

Jedno z nejznámějších rozdělení je podle tzv. *Chomského hierarchie*:

- **Gramatiky typu 0** (frázové/neomezené gramatiky)
  Zahrnují všechny formální gramatiky.
  Model pro zpracování se nazývá *Turingův stroj*.
  Tvoří třídu *rekurzivně spočetných jazyků*, zkratka **RE**.

- **Gramatiky typu 1** (kontextové gramatiky)
  Tyto gramatiky se skládají z pravidel typu $\alpha A \beta \to \alpha \gamma \beta$, kde A je neterminál a α, β, γ jsou řetězce terminálů i neterminálů, přičemž γ je neprázdný.
  Model pro zpracování se nazývá *lineárně ohraničený Turingův stroj*.
  Tvoří třídu *kontextových jazyků*, zkratka **CS**.

- **Gramatiky typu 2** (bezkontextové gramatiky)
  Skládají se z pravidel typu $A \to \gamma$, kde A je neterminál a γ řetězec terminálů a neterminálů.
  Model pro zpracování se nazývá *nedeterministický zásobníkový automat*.
  Tvoří třídu *bezkontextových jazyků*, zkratka **CF**.

- **Gramatiky typu 3** (regulární gramatiky)
  Skládají se z pravidel typu $A \to B$ a $A \to aB$, kde A, B jsou neterminály a a je

terminál.
Model pro zpracování se nazývá *konečný automat*.
Tvoří třídu *regulárních jazyků*, zkratka **REG**.

Vyjadřovací síla jazyka

Čím větší má jazyk vyjadřovací sílu, tím detailnější omezení je jeho gramatika schopna klást na přijímané řetězce. Jsme tedy schopni jemněji rozlišovat, které řetězce patří do jazyka, a které ne.

V *Chomského hierarchii* jsou jazyky uspořádány tak, že slabší jazyk je vždy podmnožinou silnějšího. Tedy například mezi bezkontextové jazyky patří i všechny regulární jazyky.

Obrázek 3.2: Schématické znázornění Chomského hieararchie

3.1 Regulární jazyky

Regulární jazyky jsou nejjednodušší formální jazyky v Chomského hierarchii. I přesto si však našly široké využití v různých oblastech informačních technologií. Využívají se např. pro pokročilé vyhledávání v textu nebo pro rozdělení programovacího jazyka na základní jednotky. V implementační části této práce jsou využity ke kontrole úrovní derivačního stromu, také proto se jimi budeme hlouběji zabývat.

Definice 3.1.1. (Regulární jazyk)
Regulární jazyk nad abecedou Σ lze definovat následovně:

- Prázdný jazyk $\emptyset$ je regulární.
- Pro každé a z Σ je $\{a\}$ regulární.
- Jestliže A a B jsou regulární jazyky, poté všechny tyto jazyky jsou také regulární: $A \cup B$ (sjednocení), AB (konkatenace) a A^* (iterace).

▲

3.1.1 Konečné automaty

Každý regulární jazyk lze zpracovávat konečným automatem a každý konečný automat lze vyjádřit regulárním jazykem. [5]

Definice 3.1.2. (Konečný automat)
Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je množina stavů
- Σ je *vstupní abeceda*
- R je množina *přechodových pravidel*
- s je *počáteční stav*
- F je množina *konečných stavů*

▲

Přechodová pravidla jsou ve tvaru $qa \rightarrow p$, kde q, p jsou stavy a a je vstupní symbol. Pravidlo nám říká, že jsme-li ve stavu q a na vstupu máme symbol a, poté může automat přejít do stavu p. Začínáme vždy v počátečním stavu a aby vstupní řetězec patřil do jazyka, musíme skončit v jednom z konečných stavů. Velkou výhodou je, že práce konečného automatu je paměťově velmi nenáročná, jelikož obsahuje pouze informaci o aktuálním stavu.

Příklad 3.1.1. Mějme konečný automat M1:

```
M1 = (
  {s, q, f},          // množina stavů
  {a, b, c},          // abeceda
  {                   // množina pravidel
    sa → q,
    qb → q,
    qc → f
  },
  s,                  // počáteční stav
  {f}                 // množina ukončujících stavů
)
```

A řetězec:

```
abbc
```

Při kontrole vstupního řetězce budeme postupovat následovně:

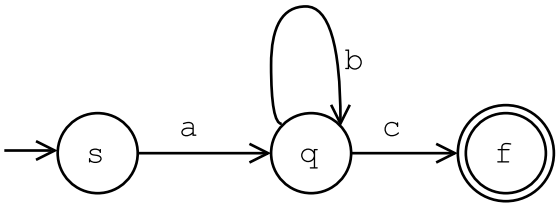
1. Nastavíme počáteční stav s
2. Vstupním symbolem je `a` - podle prvního pravidla přejdeme do stavu q
3. Vstupním symbolem je `b` - zůstáváme ve stavu q (pr. 2)
4. Vstupním symbolem je `b` - zůstáváme ve stavu q (pr. 2)
5. Vstupním symbolem je `c` - přejdeme do stavu f (pr. 3)
6. Jsme na konci řetězce - zkontrolujeme, zdali jsme v konečném stavu - řetězec byl automatem přijat, takže řetězec patří do jazyka generovaného automatem.

Během činnosti konečného automatu mohou nastat tyto chyby:

- vstupní symbol nepatří do abecedy
- neexistuje pravidlo pro vstupní symbol a aktuální stav
- po přečtení posledního znaku se nenacházíme v konečném stavu

Ve všech těchto případech není vstupní řetězec přijat konečným automatem, tudíž nepatří do jazyka generovaného automatem.

Tento konečný automat lze také zobrazit pomocí následujícího grafu:



■

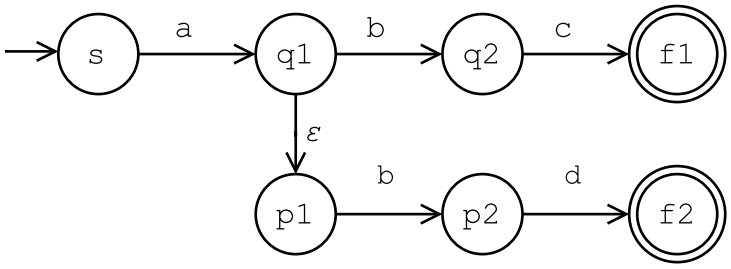
Uveďme další příklad, který už nebude tak jednoduchý:

Příklad 3.1.2. Mějme konečný automat M2:

```
M2 = (
  {s, q1, q2, p1, p2, f1, f2},
  {a, b, c, d},
  {
    sa → q1,
    q1b → q2,
    q2c → f1,
    q1ε → p1,
    p1b → p2,
    p2d → f2
  },
  s1,
  {f1}
)
```

Za pozornost stojí hlavně čtvrté pravidlo s ε přechodem. Toto pravidlo značí, že lze bez přijetí jakéhokoliv znaku přejít z jednoho stavu do druhého. Tyto pravidla se bohužel mohou v obecných konečných automatech vyskytovat a způsobují potíže při zpracovávání automatu, jak bude vysvětleno dále.

Pro větší názornost budeme nyní pracovat se schématem automatu:



Pokud se při zpracování ocitneme ve stavu $q1$ a vstupním symbolem bude `b`, není jasné, jestli máme použít 2. nebo 4. pravidlo. Museli bychom vyzkoušet obě cesty a až zpětně bychom zjistili, která možnost byla správná. Tomuto jevu se v teorii formálních jazyků říká *nedeterminismus* a setkáme se s ním ještě několikrát.

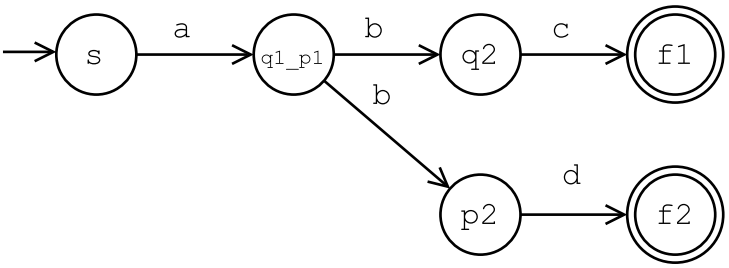
■

Nutno ale podotknout, že v tomto případě *nedeterminismus* neznamená, že bychom nebyli schopni určit, jestli řetězec patří do daného jazyka. Ve skutečnosti totiž můžeme vyzkoušet všechna možná pravidla a zjistit, jestli z nich některé povede k úspěchu. Slepé zkoušení pravidel však vede k velkému zpomalení vyhodnocování, může totiž dojít k tomu, že se program bude větvit opakovaně a délka zpracování bude neúnosná. Požadavek na striktní determinismus je tedy veden snahou o co nejrychlejší zpracování.

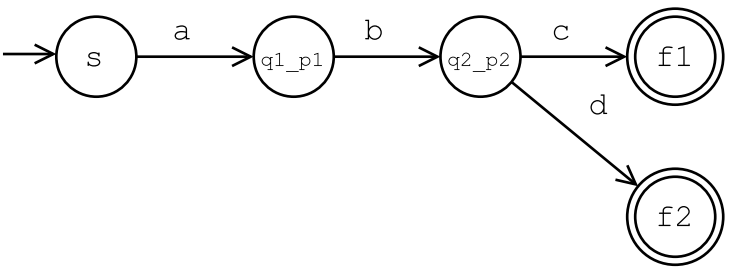
3.1.2 Determinizace konečného automatu

Z výše uvedených odstavců je zjevné, že je lepší se preventivně nedeterminismu zbavit. Nejprve ukážeme demonstraci na tomto konkrétním příkladě a poté uvedeme obecný algoritmus.

Abychom se zbavili ε přechodů, musíme vytvořit nový automat, který ale generuje stejný jazyk (přijímá stejné řetězce) jako ten původní. Takové automaty se označují jako *ekvivalentní*. Protože můžeme z přechodu $q1$ kdykoliv přejít do $q2$, intuitivním řešením je oba stavy spojit do jednoho. Pro zachování *ekvivalence* musíme do nového stavu přidat všechna pravidla, která vycházela z původních dvou stavů. Nový automat bude vypadat takto:



Pokud si nové schéma dobře prohlédneme, odhalíme další zádrhel, který se nám objevil v novém pravidle $q1\_p1$. Pokud je v tomto stavu na vstupu symbol `b`, nevíme, které pravidlo použít a máme tu opět *nedeterminismus*. I tento problém lze naštěstí vyřešit obdobným způsobem:



Tento automat můžeme označit jako *deterministický* a lze jej s klidem implementovat.

V tomto konkrétním případě jsme si pomohli intuicí, uveďme ale obecné algoritmy:

Algoritmus 3.1: Stavy dostupné bez čtení ze stavů E (*ε-closure(E)*)

Input: Konečný automat $M = (Q, \Sigma, R, s, F)$ a $E \subseteq Q$
Output: *ε-closure(E)*

```
1 begin
2     ε-closure(E) := E;
3     repeat
4         ε-closure(E) := ε-closure(E) ∪ {p|q → p ∈ R and q ∈ ε-closure(E)}
5     until ε-closure(E) nebyl změněn;
```

Algoritmus 3.2: Odstranění ε pravidel

Input: Konečný automat $I = (Q_I, \Sigma_I, R_I, s_I, F_I)$ a $E \subseteq Q$
Output: Konečný automat bez ε pravidel O, ekvivalentní s I

```
1 begin
2     Q_O := Q_I;
3     Σ_O := Σ_I;
4     s_O := s_I;
5     F_O := {q|q ∈ Q_I, ε-closure(q) ∩ F_I ≠ ∅};
6     R_O := {qa → p|q ∈ Q_I, and ∈ Σ_I, oa → p ∈ R_I pro všechna o ∈ ε-closure(q) v I}
```

Algoritmus 3.3: Odstranění nedeterminismu

Input: Konečný automat bez ε-přechodů $M = (Q, \Sigma, R, s, F)$
Output: Deterministický KA: $D = (Q_D, \Sigma, R_D, s_D, F_D)$ ekvivalentní s M

begin
  $s_D := \{s\}$;
  $Q_{new} := \{s_D\}$;
  $R_D := \emptyset$;
  $Q_D := \emptyset$;
  $F_D := \emptyset$;
  repeat
    nechť $Q' \in Q_{new}$;
    $Q_{new} := Q_{new} - \{Q'\}$;
    $Q_D := Q_D \cup Q'$;
    forall the $a \in \Sigma$ **do**
      $Q'' := \{q | p \in Q', pa \to q \in R\}$;
      if $Q'' \neq \emptyset$ **then**
        $R_D := R_D \cup \{Q'a \to Q''\}$;
      if $Q'' \notin Q_D \cup \{\emptyset\}$ **then**
        $Q_{new} := Q_{new} \cup \{Q''\}$;
    if $Q' \cap F \neq \emptyset$ **then**
      $F_D := F_D \cup \{Q'\}$
  until $Q_{new} = \emptyset$;

Tyto algoritmy jsou definovány pro jakýkoliv konečný automat, tedy jakýkoliv KA lze převést na deterministický KA [5, str. 39]. Z toho vyplývá, že jakýkoliv regulární jazyk lze zpracovávat deterministicky, což u jazyků z ostatních tříd *Chomského hierarchie* neplatí.

Existují také další transformace konečného automatu jako např. odstranění nedostupných stavů nebo minimalizace. V této práci však nebudou využívány, a proto zde nejsou více rozebírány.

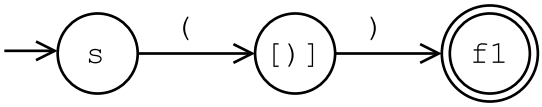
Na co konečné automaty nestačí

Příklad 3.1.3. Řekněme, že chceme konečným automatem kontrolovat, jestli matematický výraz obsahuje stejné množství otevíracích závorek jako uzavíracích a jestli jsou ve správném pořadí.
Příklady řetězců:

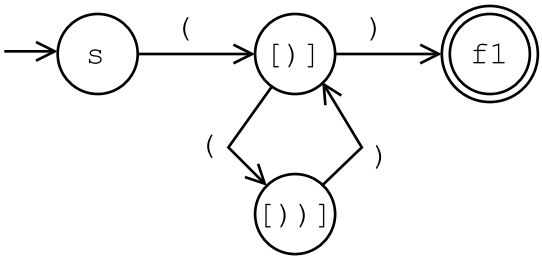
- `(())` - v pořádku
- `(()())` - v pořádku
- `(()` - špatně
- `)(` - špatně

Příklad si zjednodušíme tak, že nebudeme uvažovat žádná čísla ani znaky uvnitř závorek.

Po přečtení prvního symbolu `(` musíme přejít do stavu, který značí „čekám jednu uzavírací závorku“ (pro větší přehlednost budeme tento stav označovat `[)]`), a když je dalším znakem `)`, přejdeme do konečného stavu. Tento konečný automat lze znázornit takto:



Tento automat bude fungovat bez problému pro řetězec `()`, my ale chceme zpracovávat i zanořené závorky a na to tento automat zatím nestačí. Pokud tedy chceme univerzálnější automat, stačí přidat další stav:



Zde již zvládneme i závorky s jedním zanořením. Problémem ovšem je, že pro každé nové zanoření musíme přidat nový stav. Kdybychom tedy chtěli zpracovávat jakýkoliv počet zanoření (tedy potenciálně ∞), musel by automat obsahovat nekonečné množství stavů. Problémem *konečného automatu* je, že může obsahovat pouze konečný počet stavů, jelikož každý musíme nejprve definovat. Zde tedy vidíme příklad jazyka, který nejde zpracovávat konečným automatem a z toho vyplývá, že není ani regulárním jazykem. Řekněme si rovnou, že *bezkontextové jazyky* tento problém řeší a k tomuto příkladu se ještě vrátíme.

■

3.2 Bezkontextové jazyky

Bezkontextový jazyk je obvykle definován bezkontextovou gramatikou. Jak jsme již uvedli v *Chomského hierarchii* (str. 6), tyto gramatiky obsahují pravidla ve tvaru $A \to \gamma$, kde A je neterminál a γ řetězec terminálů a neterminálů.

3.2.1 Bezkontextové gramatiky

Pokračujme nyní v příkladu 3.1.3 a ukažme si, jak ho lze vyjádřit pomocí bezkontextové gramatiky.

Chceme tedy vyjádřit výraz tvořený závorkami pomocí gramatických pravidel. Označme dvojici závorek jako výraz = neterminál E. Gramatické pravidlo tedy bude vypadat takto:

$$E \to ()$$

Nyní bychom ale chtěli vyjádřit, že uvnitř závorek může být další výraz, to lze udělat tímto rekurzivním způsobem:

$$E \to (E)$$

Zkusme nyní rozgenerovávat výraz E, tak jak to bylo naznačeno na Obr. 3.1, tedy pomocí přepisování neterminálů:

```
1.         E
2.        (E)
3.       ((E))
4.      (((E)))
5.        ...
```

Vidíme, že zanořování, které nám dělalo problémy u regulárních jazyků, zde vyjádříme bez problému, ještě by to však chtělo několik vylepšení. Můžeme si všimnout, že rozgenerovávání by se vlastně mělo provádět do nekonečna, protože neterminálu E se nyní nelze zbavit. To lze vyřešit přidáním tzv. ε-pravidla:

$$E \to \varepsilon$$

To nám říká, že neterminál E je možno kdykoliv vymazat. Dále bychom ještě chtěli, aby se za závorkou mohla vyskytovat další závorka, např. `(()())`. Stačí přidat do výrazu další rekurzi:

$$E \to (E)E$$

Všimněme si ještě, že začínáme od symbolu E, který lze vymazat pomocí ε-pravidla, přijímáme tedy i prázdný řetězec. Pokud bychom chtěli vyjádřit, že celý výraz musí být alespoň v jedněch závorkách, lze to udělat přidáním speciálního počátečního pravidla:

$$S \to (E)$$

Nyní tedy budeme začínat od symbolu S. Označme si tuto gramatiku jako G a vyjádřeme ji formálně:

```
G = (
  {S, E},              // množina neterminálů
  {(, )},              // množina terminálů
  {                    // množina pravidel
    S → (E),
    E → (E)E,
    E → ε
  },
  S                    // počáteční symbol
)
```

Je zřejmé, že bezkontextovými jazyky jsme schopni popsat mnohem složitější jazyky než regulárními jazyky. Daní za větší *vyjadřovací sílu* je ale znatelně složitější zpracování těchto jazyků.

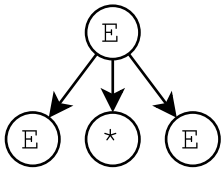
3.2.2 Derivační strom

Při aplikaci gramatiky na řetězec jde vlastně o ověření, jestli lze z počátečního symbolu postupnou derivací (aplikací pravidel gramatiky) získat daný řetězec. Mějme například gramatiku:

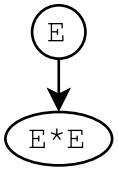
```
G = (
  {E},
  {*, +, a},
  {
    E → E*E,
```

```
      E → E+E,
      E → a
    },
    E
)
```

Pravidla této gramatiky lze vyjádřit pomocí elementárního stromu, kde levá strana je kořen a pravá strana představuje jeho potomky. Např. první pravidlo z gramatiky G lze znázornit stromem takto:



Pro menší velikost grafu budeme ale pravidla zjednodušeně znázorňovat takto:



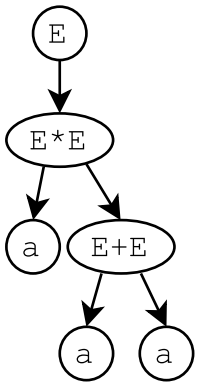
Pro příklad postupné derivace mějme řetězec s:

```
a * a + a
```

Nyní zkusíme postupně aplikovat pravidla na počáteční symbol, tak abychom získali řetězec s (v komentáři jsou uvedena pravidla, která byla použita):

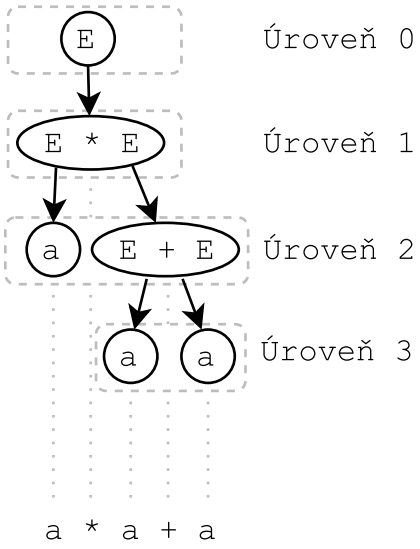
```
 E
E*E                    // E → E*E[1]
a*E                    // E → a
a*E+E                  // E → E+E
a*a+E                  // E → a
a*a+a                  // E → a
```

Touto posloupností derivací jsme byli schopni dosáhnout kontrolovaného řetězce s, daný řetězec tedy patří do gramatiky. Výše zobrazené derivace lze zobrazit i jinak, a to pomocí tzv. *derivačního stromu*:



[1] Zde bychom mohli použíti pravidlo $E \to E + E$ a dostali bychom odlišný strom (viz Sekce 3.2.3)

Pro větší názornost ukažme ještě stejný strom s přiřazenými symboly k původnímu řetězci a vyznačenými jednotlivými úrovněmi.



Definice 3.2.1. (Derivační strom). [5, str. 92]
Nechť $G = (\Sigma, R)$ je bezkontextová gramatika.

1. Pro l: $A \to x \in R, A\langle x \rangle$ je strom pravidla, které reprezentuje l.
2. Derivační strom reprezentující derivace v G je definován rekurzivně:
   (a) Strom s jedním uzlem X je derivační strom odpovídající $X \Rightarrow^0 X$ v G, kde $X \in \Sigma$.
   (b) Nechť d je derivační strom reprezentující $A \Rightarrow^0 uBv\ [\rho]$ s hranicí$(d) = uBv$, a nechť $l : B \to z \in R$. Derivační strom, který reprezentuje

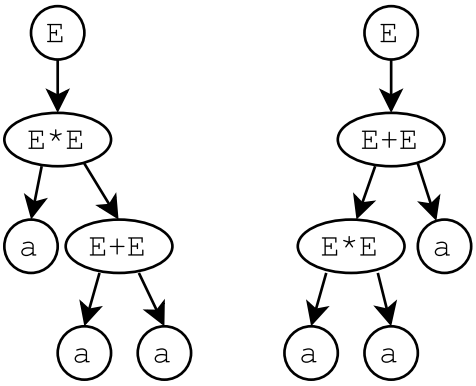
$$\begin{aligned} A &\Rightarrow^* uBv[\rho] \\ &\Rightarrow\ uzv[l] \end{aligned}$$

   je získán nahrazením $(|u|+1)$-tého listu v d, B, stromem pravidla odpovídajícího l, $B\langle z \rangle$
3. Derivační strom v G je jakékoliv t, pro které existuje derivace odpovídající t (viz 2.).

▲

3.2.3 Nedeterminismus

U příkladu z minulé sekce (3.2.2) si můžeme všimnout, že při konstrukci derivačního stromu máme pro jeden řetězec možnost sestrojit dva různé stromy:



Vidíme, že generovaný řetězec je stejný, ale stromy jsou odlišné. Takováto gramatika je označována jako *nedeterministická* a způsobuje problémy při zpracování, protože v rozhodné chvíli nevíme, které pravidlo použít.

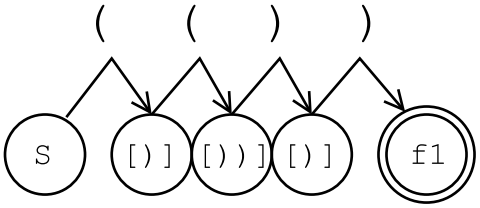
Protože determinismus je při implementaci zásadní, rozdělujeme bezkontextové gramatiky na deterministické a nedeterministické (podobně i zásobníkové automaty).

3.2.4 Zásobníkové automaty

Zásobníkový automat rozšiřuje *konečný automat* o zásobník, kam lze ukládat symboly (terminály i neterminály).

Pro rozhodnutí, jaké pravidlo použít, využívá vedle vstupního symbolu a stavu i symbol na vrcholu zásobníku. V rámci vykonání přechodu lze zároveň manipulovat se zásobníkem.

Vraťme se opět k příkladu 3.1.3 a všimněme si stavů vyjadřujících zanoření. Značili jsme je jako počet uzavíracích závorek, které jsou ještě potřeba, aby byl výraz platný. Pokud jsme narazili na otevírací závorku, přešli jsme do stavu „o jedna více závorek“, v případě uzavírací závorky do „o jedna méně závorek“. Přechody mezi stavy jsou znázorněny na následujícím jednoduchém řetězci:



Přidávání a odebírání závorek nám může připomínat zásobník. Zásobníkový automat dokáže tyto stavy vyjádřit zásobníkem, a potom není nutné všechny tyto stavy definovat.

Definice 3.2.2. (Zásobníkový automat)
Zásobníkový automat je sedmice $M = (Q, \Sigma, \Gamma, R, q_0, Z_0, F)$, kde:

- Q je konečná množina stavů

- Σ je vstupní abeceda
- Γ je konečná abeceda zásobníku
- $R \subseteq (\Gamma \times Q \times (\Sigma \cup \{\varepsilon\})) \times (\Gamma^* \cup Q)$ je konečná množina binárních relací (pravidel)
- $q_0 \in Q$ je počáteční stav
- $Z_0 \in \Gamma$ popisuje počáteční symboly na zásobníku
- $F \subseteq Q$ je množina konečných stavů

Pravidla R zásobníkového automatu jsou ve tvaru $(q_1, a, z_1) \rightarrow (q_2, \gamma)$, kde:

- q_1 je výchozí stav
- a je symbol na vstupu
- z_1 je symbol na vrcholu zásobníku
- q_2 je výstupní stav
- γ je řetězec, který se má vložit na zásobník

▲

Zásobníkovým automatem lze zpracovávat deterministické bezkontextové jazyky. V případě nedeterministických musíme nějak blíže specifikovat, který z možných stromů chceme.

3.3 Zpracování bezkontextových jazyků

Při zpracování bezkontextového jazyka je hlavní problematikou sestavování konfigurace zásobníkového automatu pro danou gramatiku. Musíme totiž gramatická (přepisovací) pravidla zanést do konfigurace automatu tak, aby bylo vždy jasné, které použít.

Poté je zde ještě otázka postupu při samotném zpracování řetězce. Při konstrukci derivačního stromu lze totiž postupovat buď od kořene (shora dolů), nebo od zkoumaného řetězce (zdola nahoru). Rozhodnout jakým způsobem postupovat není úplně jednoznačné, což je vidět i na široce používaných analyzátorech programovacích jazyků, kdy se tato technika liší projekt od projektu.

V případě tohoto projektu je tomu nejinak, a proto v této části budeme rozebírat obě alternativy, aby byly zřejmé jejich výhody i nevýhody.

3.3.1 Syntaktická analýza shora dolů

Nejpoužívanějším zástupcem této skupiny je *LL syntaktická analýza*, která analyzuje vstup zleva doprava a konstruuje nejlevější derivaci. Tato syntaktická analýza umožňuje zpracovávat pouze *LL gramatiky*, které jsou podmnožinou deterministických bezkontextových gramatik.

Hlavní součástí je *LL tabulka*, která nám na základě vstupního symbolu a symbolu na zásobníku určí pravidlo, které se má použít. Následující algoritmy slouží k její konstrukci.

Množina $Empty(X)$ nám říká, jestli lze symbol X odstranit:

Algoritmus 3.4: $Empty(X)$

Input: Gramatika $G = (N, \Sigma, P, S)$
Output: $Empty(X)$ pro každý symbol $X \in N \cup \Sigma$

```
1  begin
2      Empty(a) := ∅ pro každé a ∈ Σ;
3      forall the A ∈ N do
4          if A → ε ∈ P then
5              Empty(A) := {ε};
6          else
7              Empty(A) := ∅;
8      repeat
9          if A → X1X2...Xn ∈ P and Empty(Xi) = ε pro všechna i = 1, ..., n then
10             Empty(A) := {ε};
11     until žádná z množin Empty nezměněna;
```

Množina $First(X)$ nám říká, které terminální symboly se mohou nacházet na začátku symbolu X:

Algoritmus 3.5: $First(X)$

Input: Gramatika $G = (N, \Sigma, P, S)$
Output: $First(X)$ pro každý symbol $X \in N \cup \Sigma$

```
1  begin
2      first(a) := {a} pro každé a ∈ Σ;
3      first(A) := ∅ pro každé A ∈ N;
4      repeat
5          if A → X1X2...Xk−1Xk...Xn ∈ P then
6              First(A) := First(A) ∪ First(X1);
7              if Empty(Xi) = {ε} pro i = 1, ..., k − 1 kde k ≤ n then
8                  First(A) := First(A) ∪ First(Xk);
9      until žádná z množin First nezměněna;
```

Množina $Empty(X_1X_2...X_n)$ nám říká, jestli lze řetězec $X_1X_2...X_n$ odstranit:

Algoritmus 3.6: $Empty(X_1X_2...X_n)$

Input: Gramatika $G = (N, \Sigma, P, S)$; $Empty(X)$ pro každé $X \in N \cup T$;
$x = X_1X_2...X_n$, kde $x \in (N \cup T)^+$
Output: $Empty(X_1X_2...X_n)$

```
1 begin
2     if Empty(X_i) = {ε} pro každé i = 1, ..., n then
3         Empty(X_1X_2...X_n) := {ε};
4     else
5         Empty(X_1X_2...X_n) := ∅;
```

Množina $First(X_1X_2...X_n)$ nám říká, které terminální symboly se mohou nacházet na začátku řetězce $X_1X_2...X_n$:

Algoritmus 3.7: $First(X_1X_2...X_n)$

Input: Gramatika $G = (N, \Sigma, P, S)$; $First(X)$ a $Empty(X)$ pro každé $X \in N \cup T$;
$x = X_1X_2...X_n$, kde $x \in (N \cup T)^+$
Output: $First(X_1X_2...X_n)$

```
1 begin
2     First(X_1X_2...X_n) := First(X_1);
3     repeat
4         if Empty(X_i) = {ε} pro i = 1, ..., k − 1 kde k ≤ n then
5             First(X_1X_2...X_n) := First(X_1X_2...X_n) ∪ First(X_k);
6     until množina First(X_1X_2...X_{k−1}X_k...X_n) nezměněna;
```

Množina $Follow(X)$ říká, které terminální symboly se mohou nacházet za symbolem X:

Algoritmus 3.8: $Follow(X)$

Input: Gramatika $G = (N, \Sigma, P, S)$
Output: $Follow(A)$ pro každý symbol $A \in N$

```
1 begin
2     Follow(S) := {$};
3     repeat
4         if A → xBy ∈ P then
5             if y ≠ ε then
6                 Follow(B) := Follow(B) ∪ First(y);
7             if Empty(y) = {ε} pro i = 1, ..., k − 1 kde k ≤ n then
8                 Follow(B) := Follow(B) ∪ Follow(A);
9     until žádná z množin Follow nezměněna;
```

Množina $Predict(A \to x)$ říká, které terminální symboly mohou být na vstupu pro pravidlo $A \to x$:

Definice 3.3.1. (Množina $Predict(A \to x)$)
Nechť $G = (N, \Sigma, P, S)$ je bezkontextová gramatika. Pro každé $A \to x \in P$ definujeme

množinu $Predict(A \to x \in P)$ takto:

Pokud $Empty(x) = \{\varepsilon\}$ **potom:**
$Predict(A \to x \in P) = First(x) \cup Follow(A)$

Jinak:
$Predict(A \to x \in P) = First(x)$

▲

Zbývá už jen naplnit LL tabulku pro funkci $\alpha(A, a)$, která nám pro neterminál a terminál určí pravidlo, které použít:

Algoritmus 3.9: $\alpha(A, a)$

Input: Gramatika $G = (N, \Sigma, P, S)$; $Predict(A \to x)$, pro každé pravidlo z P
Output: $\alpha(A, a)$, pro všechny platné kombinace (A, a)

```
1 begin
2     forall the p : A → a ∈ P do
3         forall the b ∈ Predict(A → a) do
4             if α(A, b) není definováno then
5                 α(A, b) := p
6             else if p ≠ α(A, b) then
7                 chyba - nejde o LL gramatiku
```

Chybový stav v posledním algoritmu nám vlastně indikuje nedeterminismus v LL tabulce - tedy že pro nějakou kombinaci $[A, a]$ by v LL tabulce bylo více záznamů, takže bychom nevěděli, které pravidlo použít.

Vraťme se nyní ke gramatice G (z kapitoly 3.2.1):

Příklad 3.3.1.

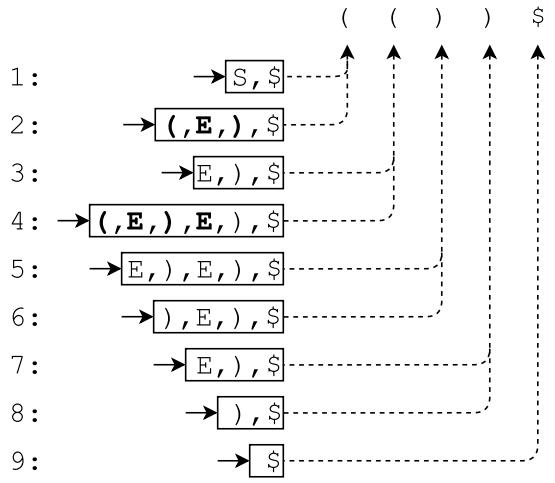
```
G = (
  {S, E},              // množina neterminálů
  {(, )},              // množina terminálů
  {                    // množina pravidel
    S → (E),           // pravidlo 1
    E → (E)E,          // pravidlo 2
    E → ε              // pravidlo 3
  },
  S                    // počáteční symbol
)
```

LL tabulku bychom podle výše uvedených algoritmů sestavili takto:

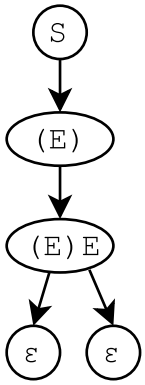
	(	)	$
S	1		
E	2	3	

Tabulka 3.1: LL tabulka gramatiky G s čísly pravidel

Nyní zkusme zpracovat řetězec `(())` gramatikou G s pomocí LL tabulky. Vlevo je znázorněný zásobník (s vrcholem) a šipky ukazují na aktuální vstupní symbol. Speciální znak `$` nám značí konec vstupu.



(a) Zpracování řetězce



(b) Derivační strom

Při zpracování jsme postupovali tak jako zásobníkový automat. Před zahájením algoritmu vložíme na zásobník symbol `$` a počáteční symbol `S`. Vysvětleme podrobně některé kroky (číslování odpovídá obrázku):

1. Na zásobníku je neterminál `S` - na základě vstupu `(` vybereme z LL tabulky pravidlo 1 a nahradíme `S` za pravou stranu pravidla
2. Na zásobníku je terminál `(` a shoduje se se vstupním symbolem - můžeme jej odstranit
3. Na zásobníku je neterminál `E` a na vstupu opět `(`, použijeme pravidlo 2
4. Na zásobníku je terminál `(` a shoduje se se vstupem - odstraníme
5. Na zásobníku je neterminál `E` a na vstupu `)` - použijeme pravidlo 3

   ...

9. Na zásobníku je terminál `$` a shoduje se se vstupem - při tomto znaku skončíme

■

Při zpracování mohou nastat následující chyby:

- Znak na vstupu není v abecedě
- V LL tabulce není pravidlo pro danou kombinaci neterminálu a terminálu
- Terminály na vstupu a vrcholu zásobníku se neshodují

Všechny tyto případy znamenají, že řetězec nepatří do gramatiky G

3.3.2 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru sestavuje derivační strom odspodu, tedy začíná neterminály a postupně se propracuje až k počátečnímu symbolu gramatiky.

LR syntaktická analýza

Pro účely této práce se budeme zabývat *LR syntaktickou analýzou*, jelikož parsery tohoto typu mohou mít až sílu ekvivalentní s *deterministickým zásobníkovým automatem*[5, str. 155], tedy představují nejsilnější nástroj v oblasti *deterministických bezkontextových jazyků*. LR syntaktická analýza generuje obrácený pravý rozbor.

Pro zpracování se opět používá *zásobníkový automat*, který je rozšířen o možnost pracovat na vrcholu zásobníku s řetězcem (ne jen s jedním symbolem). Toto rozšíření je nutné pro aplikování pravidel - tedy nahrazení řetězce na pravé straně za symbol na levé straně pravidla (postupujeme opačně než u LL).

Při analýze řetězce dáváme postupně příchozí symboly na zásobník (operace *shift*) a v určité chvíli použijeme pravidlo a nahradíme symboly na vrcholu zásobníku za jeho levou stranu (operace *reduce*). Kterou operaci v dané chvíli provést, nám určuje *LR tabulka*, jejíž konstrukcí se budeme dále zabývat.

LR tabulka

LR tabulka je poněkud složitější než *LL tabulka*. Obsahuje 2 části:

- Akční část je definována jako $\alpha(q, a)$, kde q je stav a a je terminál.
  Určuje, jakou operaci provést a k tomu informaci, do jakého stavu přejít (u op. shift) nebo jaké pravidlo použít (op. reduce), a také speciální ukončovací symbol
- Přechodová část je definována jako $\beta(q, A)$, kde q je stav a A je neterminál.
  Určuje, do jakého stavu přejít po aplikaci pravidla.

Stav zde vyjadřuje možnosti, které vyplývají ze znaků naposledy přečtených. Pro každý stav existuje množina pravidel, jejichž pravá strana prozatím vyhovuje přečteným symbolům. Při znázorňování množin pravidel budeme používat znak $\bullet$ znázorňující aktuální pozici v pravidle.

Algoritmus $Closure(I)$ je definován pro pravidlo s určenou pozicí a generuje z něj stavovou skupinu:

Algoritmus 3.10: $Closure(I)$

Input: Gramatika $G = (N, \Sigma, P, S)$; Položka I
Output: $Closure(I)$

```
1 begin
2     Closure(I) := {I};
3     repeat
4         if A → y•Bz ∈ Closure(I) and B → x ∈ P then
5             Closure(I) := Closure(I) ∪ B → •x;
6     until množina Closure(I) nezměněna;
```

Následující algoritmus vytvoří množinu Θ_G všech stavových množin pro gramatiku G rozšířenou o pravidlo $S' \to S$, kde S je počáteční symbol:

Algoritmus 3.11: Θ_G

Input: Rozšířená gramatika $G = (N, \Sigma, P, S')$
Output: Θ_G

```
1 begin
2     Θ_G := {Closure(S' → •S)};
3     forall the I ∈ Θ_G and U ∈ N ∪ Σ do
4         if Θ_U(I) ≠ ∅ then
5             přidej Θ_U(I) do Θ_G;
```

Nyní sestrojíme LR tabulku pomocí SLR algoritmu. Budeme zde potřebovat také algoritmus 3.8 ($Follow$). Operaci shift znázorňujeme jako s, reduce jako r. Ukončovací symbol označme jako `a` (Accept).

Algoritmus 3.12: SLR

Input: Rozšířená gramatika $G = (N, \Sigma, P, S')$; Θ_G; $Follow(A)$ pro všechna $A \in N$
Output: LR tabulka pro G (α = akční č., β = přechodová č.)

```
1  begin
2      forall the x ∈ Θ_G do
3          forall the I ∈ x do
4              switch I do
5                  case I = A → y•Xz, kde X ∈ N:
6                      β[x, X] := Θ_X(x) ;                          // β část
7                  case I = A → y•Xz, kde X ∈ Σ:
8                      α[x, X] := sΘ_X(x) ;                         // operace shift
9                  case I = S' → S•:
10                     α[x, $] := a ;                               // úspěšný konec
11                 case A → y• (A ≠ S'):
12                     forall the a ∈ Follow(A) do
13                         α[x, a] := rp, kde p je číslo pravidla A → y ;   // reduce
```

Syntaktická analýza využívající LR tabulku vypadá takto:

Algoritmus 3.13: LR syntaktická analýza

Input: LR tabulka pro $G = (N, \Sigma, P, S)$ a řetězec $x \in T^*$
Output: Pravý rozbor x, pokud $x \in L(G)$, jinak chyba

```
1  begin
2      Vlož ⟨$, q0⟩ na zásobník;
3      stav := q0;
4      repeat
5          a = aktuální znak na vstupu;
6          switch α[stav, a] do
7              case sq:
8                  push(⟨a, q⟩);
9                  přečti další znak a ze vstupu;
10                 stav := q;
11             case rp
12                 if p: A → X1X2...Xn ∈ P and ⟨?, q⟩⟨X1, ?⟩⟨X2, ?⟩...⟨Xn, ?⟩ je na
                   vrcholu zásobníku then
13                     stav := β[q, A];
14                     zaměň na zásobníku ⟨X1, ?⟩⟨X2, ?⟩...⟨Xn, ?⟩ za ⟨A, stav⟩;
15                     zapiš p na výstup;
16                 else
17                     chyba
18             case a
19                 úspěch
20             case nedefinováno
21                 chyba
22     until úspěch or chyba;
```

Příklad 3.3.2. Vezměme gramatiku G z kapitoly 3.2.2:

```
G = (
  {E},
  {*, +, a},
  {
    S' → E,              // pravidlo 0 (přidané)
    E → E*E,             // pravidlo 1
    E → E+E,             // pravidlo 2
    E → a                // pravidlo 3
  },
  E
)
```

Stavové skupiny Θ_G z algoritmu 3.11 pro tuto gramatiku vypadají takto:

```
0: S' -> •E, E -> •E*E, E -> •E+E, E -> •a
1: S' -> E•, E -> E•*E, E -> E•+E
2: E -> a•
3: E -> E*•E, E -> •E*E, E -> •E+E, E -> •a
4: E -> E+•E, E -> •E*E, E -> •E+E, E -> •a
```

```
5: E -> E*E•, E -> E•*E, E -> E•+E
6: E -> E+E•, E -> E•*E, E -> E•+E
```

Zde je LR tabulka:

	α				β	
	*	+	a	$	S'	E
0			s1			2
1	r3	r3		r3		
2	s3	s4		*a*		
3			s1			5
4			s1			6
5	s3 r1	s4 r1		r1		
6	s3 r2	s4 r2		r2		

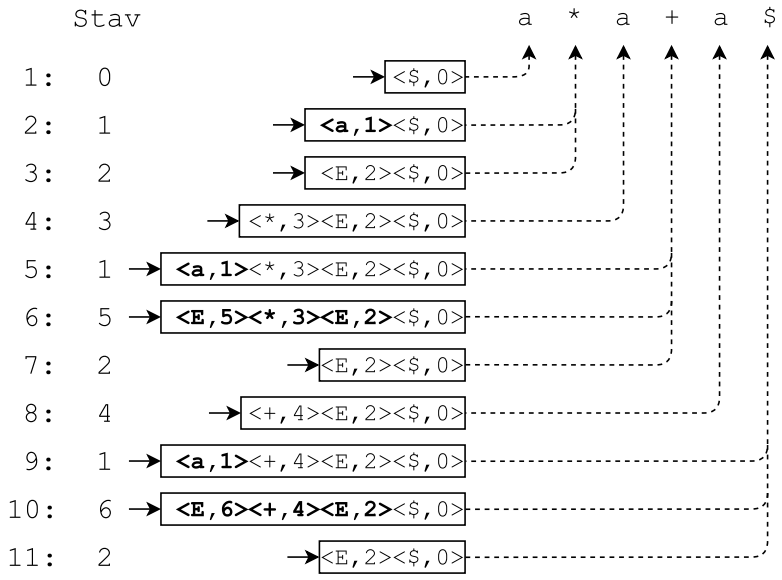
Všimněme si, že v některých polích tabulky máme 2 položky. Tyto případy se nazývají *shift-reduce* konflikty a jsou projevem toho, že je tato gramatika nedeterministická (viz sekce 3.2.3). Výhodou LR gramatiky je to, že ji lze lehce rozšířit o priority operátorů a podle nich rozhodnout, jestli provést operace *shift* nebo *reduce*. Priority pro tuto gramatiku mohou vypadat následovně:

```
left: *
left: +
```

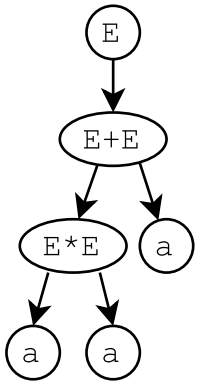
V tomto případě má tedy operátor `*` vyšší prioritu a oba operátory jsou vyhodnocovány zleva. Ještě před zpracováním řetězce můžeme tedy vyřešit konflikty v LR tabulce tak, že porovnáme prioritu operátoru v *reduce* pravidle se vstupním symbolem a podle priorit nastavíme buď *shift* (vyšší priorita vstupního symbolu nebo pravá asociativita), nebo *reduce* (nižší priorita vstupního symbolu nebo levá asociativita). Tabulka po vyřešení konfliktů vypadá takto:

	α				β	
	*	+	a	$	S'	E
0			s1			2
1	r3	r3		r3		
2	s3	s4		*a*		
3			s1			5
4			s1			6
5	r1	r1		r1		
6	s3	r2		r2		

Ukažme si zpracování řetězce `a * a + a`:



(c) Zpracování řetězce



(d) Derivační strom

Ve výše zobrazeném diagramu postupujeme podle algoritmu 3.13, vysvětleme některé kroky:

1. Jsme ve stavu 0 a na vstupu máme `a` - podle LR tabulky přejdeme do stavu 1 a provedeme operaci *shift* (vložíme `a` na zásobník s aktuálním stavem).

2. Jsme ve stavu 1 a na vstupu máme `*` - podle LR tabulky provedeme operaci *reduce* podle pravidla 3 - poslední stav za nahrazovanou částí na zásobníku je 0 a levá strana pravidla je `E` - podle beta části LR tabulky přejdeme do stavu 2 (vložíme na zásobník i s neterminálem `E`).

3. Jsme ve stavu 2 a na vstupu máme `*` - *shift* 3.

4. Jsme ve stavu 3 a na vstupu máme `a` - *shift* 1.

5. Jsme ve stavu 1 a na vstupu máme `+` - *reduce* 3 - poslední stav je 3 a levá strana je `E` - přejdeme do stavu 5.

6. Jsme ve stavu 5 a na vstupu máme `+` - (zde je konflikt vyřešený prioritou operátorů) provedeme operaci *reduce* 1 - poslední stav je 0 a levá strana je `E` - přejdeme do stavu 2.

   ...

11. Jsme ve stavu 2 a na vstupu máme `$` - v LR tabulce je a (Accept) - můžeme skončit a řetězec prohlásit za přijatý.

■

Je vidět, že pomocí LR syntaktické analýzy jsme schopni zpracovávat deterministické gramatiky a pokud přidáme priority, i některé nedeterministické. Všimněme si, jak je sestavován derivační strom - jdeme odspodu a vytváříme několik podstromů, teprve v posledním kroku je spojíme do jednoho.

Kapitola 4

Gramatiky řízené stromy

Pro dosažení větší síly syntaktického analyzátoru bývají zásobníkové automaty rozšiřovány o další funkce (tím se odchylují od původního matematického modelu). V této kapitole se budeme zabývat jazyky řízenými stromy a rozšířením zásobníkového automatu, aby byl schopen takové jazyky zpracovávat.

4.1 Omezení úrovní derivačního stromu

Ve své práci jsem se rozhodl pro techniku omezování derivačního stromu pomocí kontroly jeho úrovní regulárním jazykem. Gramatika jazyka řízeného stromem se tedy skládá ze 2 částí a značíme ji jako (G, R), kde:

- G je bezkontextová *řízená gramatika*, slouží ke konstrukci derivačního stromu pro daný řetězec
- R je *kontrolní jazyk* (budeme využívat regulární jazyk), slouží pro kontrolu úrovní derivačního stromu

Vyjádřeno formálně:

Definice 4.1.1. (Gramatiky řízené stromy)
Gramatika řízená stromem je dvojice (G, R), kde $G = (V, T, P, S)$ je *řízená gramatika* a $R \subseteq V^*$ *kontrolní jazyk.* ▲

Definice 4.1.2. (Jazyky řízené stromy)
Nechť (G, R) je gramatika řízená stromem. Jazyk generovaný (G, R) je značen jako $L(G, R)$ a je definován jako:

$L(G, R) = \{x : x \in L(G, R)$ a existuje derivační strom t pro každé x v G takový, že každé slovo, získané konkatenací všech symbolů na kterékoliv úrovni t (kromě poslední) zleva doprava, patří do $R\}$.

▲

Takto specifikované jazyky mají vyjadřovací sílu *rekurzivně vyčíslitelných jazyků.*[4] Ukážeme si ovšem, že pokud se pokusíme sestrojit analyzátor takového jazyka, narazíme na řadu potíží, které výslednou vyjadřovací sílu citelně sníží.

Pro lepší porozumění následuje podrobný praktický příklad, jak lze ověřovat příslušnost řetězce do dané gramatiky řízené stromem.

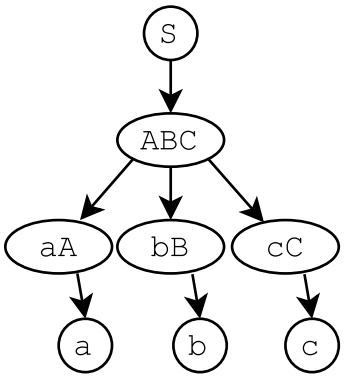
Příklad 4.1.1. Mějme stromem řízený jazyk $L(G, R)$, kde:

```
G = (
  {S, A, B, C, a, b, c},
  {a, b, c},
  {
    S → ABC,
    A → aA,
    A → a,
    B → bB,
    B → b,
    C → cC,
    C → c
  },
  S
),
R = {S, ABC, aAbBcC}
```

A řetězec:

```
aabbcc
```

Nejprve sestavíme derivační strom pro daný řetězec, poté projdeme všechny jeho úrovně (kromě poslední, viz Definice 4.1.2) a ověříme, že patří do kontrolního jazyka R.

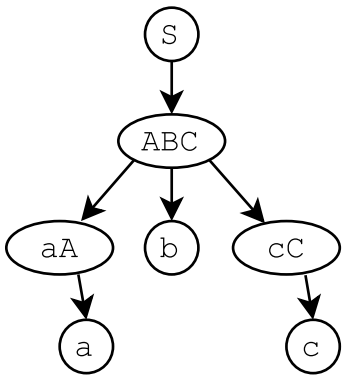


S ✓
ABC ✓
aAbBcC ✓

Z grafu je zřejmé, že v tomto případě řetězec patří do jazyka L. Zkusme však ještě jiný případ pro tento řetězec:

```
aabcc
```

A jemu odpovídající derivační strom:



S ✓
ABC ✓
aAbcC ✗

V tomto případě úroveň 2 derivačního stromu nepatří do kontrolního jazyka R, proto tento řetězec nepatří do jazyka L. ■

Ukázali jsme si, že samotná kontrola úrovní derivačního stromu není nijak velkým problémem. Potřebujeme však nejdříve derivační strom zkonstruovat.

Sestavení derivačního stromu

Zásadním problémem při konstrukci derivačního stromu je nedeterminismus, který vede k tomu, že máme pro jeden řetězec více derivačních stromů. Jelikož kontrola úrovní stromu rozhoduje o příslušnosti řetězce do jazyka, mohlo by se stát, že některé derivační stromy by kontrolou prošly, a jiné ne. V tomto případě bychom museli sestrojit všechny derivační stromy (pomocí nedeterministického syntaktického analyzátoru) a poté u všech zkontrolovat úrovně. Nedeterministické analyzátory jsou ovšem řádově pomalejší než deterministické, proto se držme těch deterministických a zkusme prozkoumat jinou možnost - tou je kontrolovat úrovně již při konstrukci derivačního stromu.

V dalších sekcích projdeme znovu dvě hlavní metody pro syntaktickou analýzu bezkontextových deterministických jazyků (rozebírané v sekci 3.3) z praktického pohledu a zamyslíme se nad tím, jak by se daly k tomuto účelu využít.

4.2 LL syntaktická analýza

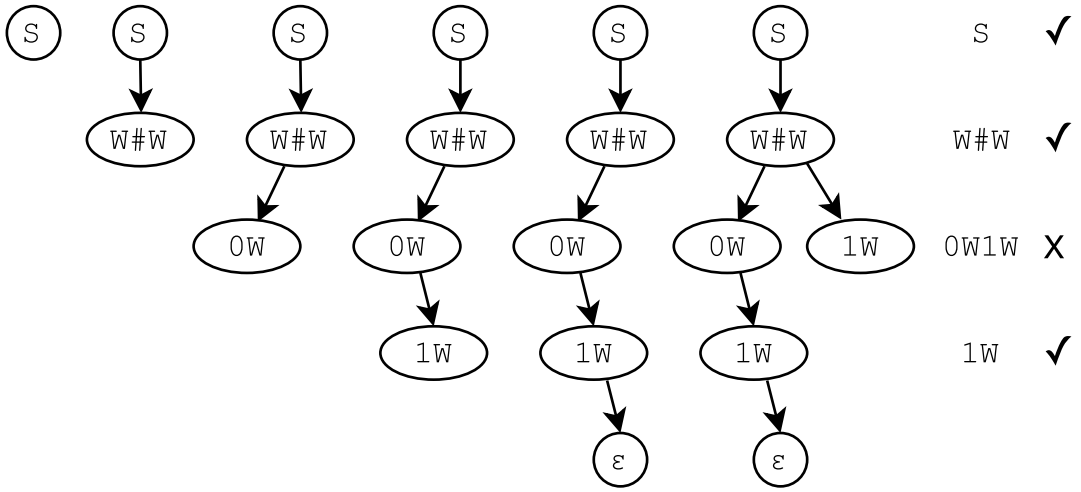
Výhodou LL syntaktické analýzy je intuitivní konstrukce derivačního stromu shora a zleva, při tomto postupu je výhodné, že vždy máme nejlevější část stromu, pokud tedy chceme kontrolovat úrovně již za běhu, můžeme to provádět bez problému, jelikož každou úroveň generujeme od nejlevějšího znaku, a můžeme ji tedy rovnou kontrolovat automatem.

Na následujícím příkladu si ukažme postup při kontrole derivačního stromu:

Příklad 4.2.1.

```
G = (
  {S, W},
  {0, 1, #},
  {
    S → W#W,
    W → 0W,
    W → 1W,
    W → ε
  },
  S
),
R = {S, W#W, 0W0W, 1W1W}
```

Pro řetězec `01#11`, vypadá LL konstrukce derivačního stromu takto:



Díky tomu, že je strom generován shora a zleva, můžeme jednoduše kontrolovat úrovně ještě před dokončením stromu. Umožňuje nám to brzké odhalení konfliktů v úrovních derivačního stromu, tedy zjištění, že řetězec nepatří do gramatiky.

■

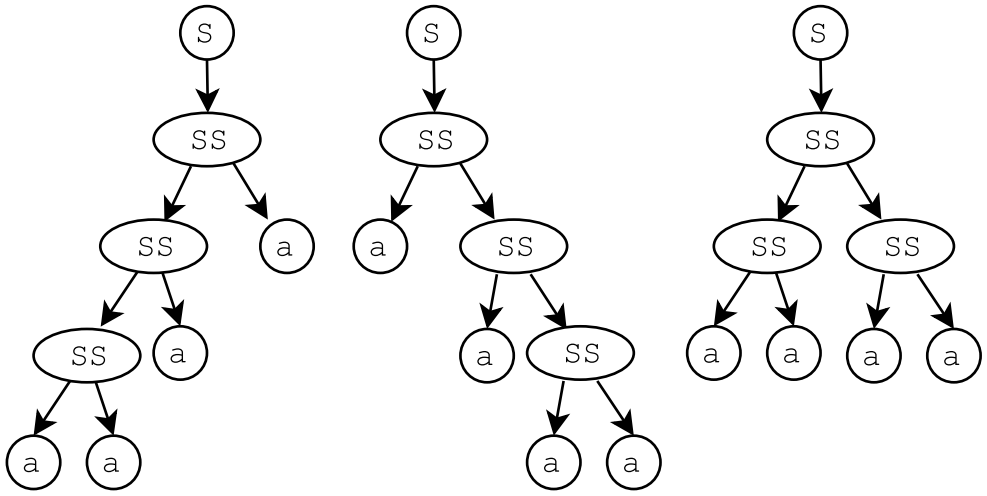
Menší komplikaci přináší to, že z definice nekontrolujeme nejspodnější úroveň stromu, protože u nekompletního stromu nejsme schopni určit, jestli je daná úroveň poslední. Spodní úroveň stromu však může obsahovat pouze terminály, a naopak ostatní úrovně vždy obsahují nějaký neterminál. Proto můžeme do automatu jednoduše přidat stavy přijímající jakoukoliv posloupnost terminálů a zkontrolovat úrovně všechny. Tato technika bude dále využívána (i u LR SA).

Nyní zkusme prozkoumat, jak bychom mohli těchto vlastností využít pro vyřešení konfliktů v *LL tabulce*. Ukažme si to na následující gramatice:

Příklad 4.2.2.

```
T = (G, R),
G = (
  {S},
  {a},
  {
    S → SS,          // pravidlo 1
    S → a            // pravidlo 2
  },
  S
),
R = {S*}
```

Gramatika T přijímá řetězec a^{2^n}. Důležité je, že *řízená gramatika* G je nedeterministická. Např. pro řetězec `aaaa` můžeme sestrojit tyto stromy:

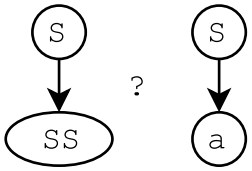


Kontrolou úrovní zjistíme, že nám vyhovuje pouze strom č. 3 - z tohoto poznatku plyne, že pokud sestrojíme špatný strom, dojdeme k mylnému závěru - že řetězec nepatří do gramatiky. Má to však i velmi pozitivní důsledek: protože kontrole úrovní vyhovuje jen jeden strom, celkově je gramatika T vlastně deterministická. Dostatečně dobrá průběžná kontrola úrovní by tedy teoreticky měla vést pouze k jednomu správnému derivačnímu stromu.

LL tabulka s konfliktem vypadá takto:

	a	$
S	0, 1	

Když zkusíme řetězec zpracovávat, už u prvního pravidla narazíme na zádrhel.



Problémem je, že oba stromy vyhovují kontrole úrovní. Nejsme tedy schopni rozhodnout, který vyloučit.

Mohli bychom také jednoduše zkusit jedno z pravidel a pokud bychom narazili na konflikt, tak se vrátit a zkusit druhé. To je již ovšem nedeterministické zpracování, a navíc se nám zde může neblaze projevit rekurze - např. pokud bychom stále aplikovali pravidlo $S \to SS$, program by stále zvětšoval strom a na konflikt v úrovních bychom nenarazili.
V tomto případě je jasné, že LL syntaktická analýza na tento problém není vhodná.

■

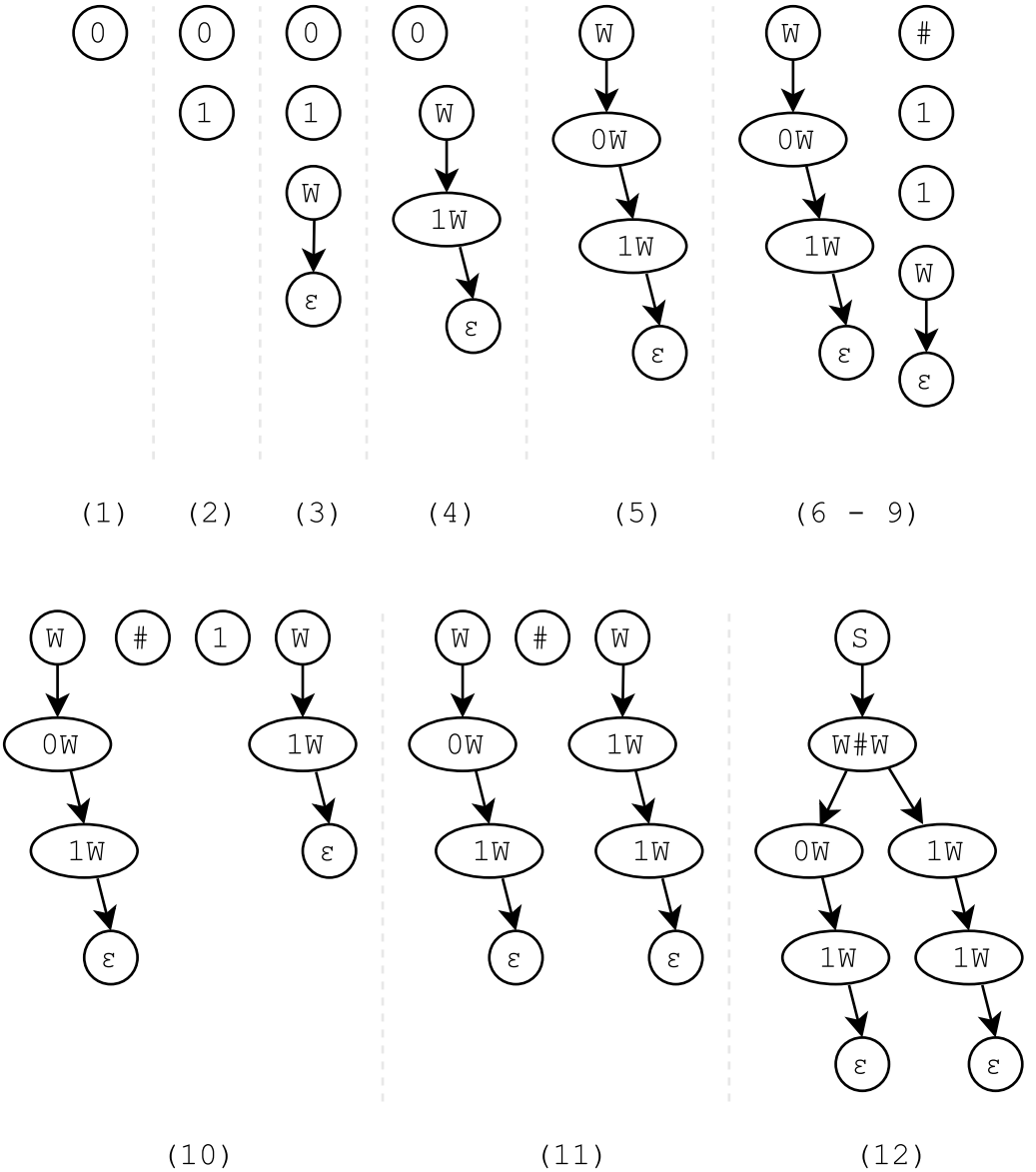
Problém LL syntaktické analýzy tkví v tom, že při aplikaci pravidla máme k dispozici pouze velmi malou část stromu vytvořeného pravidlem, a je tedy velmi pravděpodobné, že nebudeme schopni určit, jestli je pravidlo správné.

Závěr

Ukázali jsme si, že výhodou LL syntaktické analýzy je možnost kontrolovat bez problému úrovně stromu již při jeho konstrukci. Jakmile je ale *řízená gramatika* nedeterministická, kontrola úrovní nám mnoho nepomůže. Navíc LL syntaktická analýza má již v základu menší vyjadřovací sílu než LR syntaktická analýza, proto se pro zpracování gramatik řízených stromy nejeví jako příliš vhodná.

4.3 LR syntaktická analýza

LR syntaktická analýza dokáže deterministicky zpracovávat více jazyků než LL a nabízí také jednoduché rozšíření o prioritu operátorů, kontrola derivačního stromu za běhu je zde však mnohem složitější, jelikož strom vytváříme zdola. Ukažme si, jak se strom vytváří pro gramatiku z příkladu 4.2.1 a řetězec `01#11`:



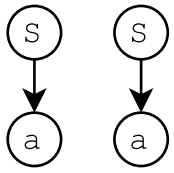
LR gramatika vytváří podstromy, které postupně spojuje do jednoho. Pokud chceme kontrolovat úrovně za běhu, lze to jednoduše provádět pouze u nejlevějšího aktuálního podstromu. Vidíme, že tímto způsobem odhalíme chybu v úrovních až v posledním kroku (12).

4.3.1 Zpracování nedeterministické řízené gramatiky

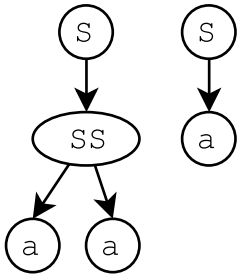
Příklad 4.3.1. Vezměme gramatiku z příkladu 4.2.2. LR tabulka pro tuto gramatiku vypadá takto:

	a	$	A
0	s2		1
1	s2	a	3
2	r2	r2	
3	s2, r1	r1	3

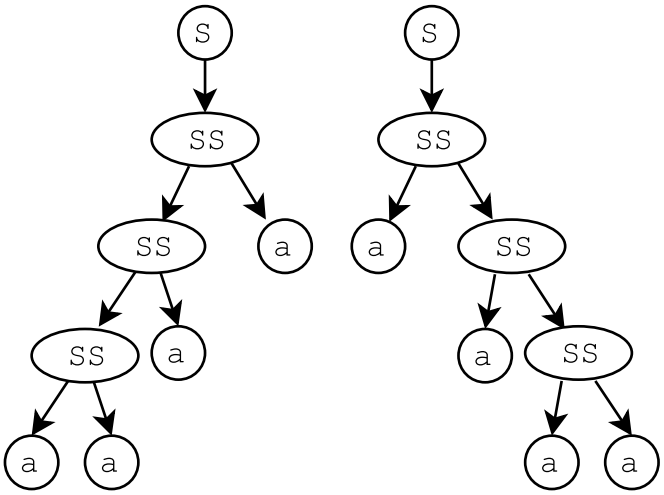
Vidíme, že tabulce je jen jediný konflikt. Při zpracování řetězce `aaaa` na něj poprvé narazíme, když máme následující 2 stromy:



Jde tedy o to, jestli stromy spojit pravidlem $1 : S \to SS$, nebo přejít na další symbol. V tomto případě chceme stromy spojit, mohli bychom tedy nastavit priority tak, aby se v případě symbolů `[S, a]` provedla operace *reduce*, tedy že `S` má větší prioritu než `a`. Při dalším konfliktu nastane tato situace:

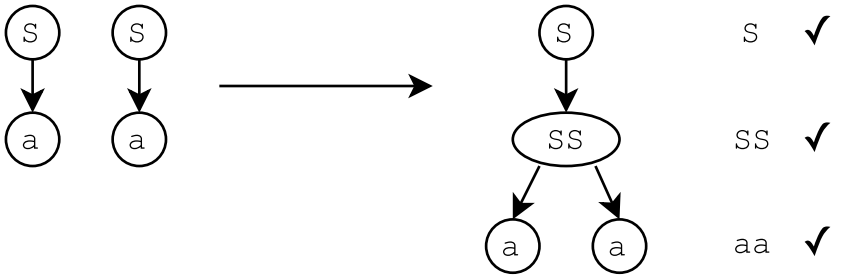


Zde ovšem s prioritami pohoříme, jelikož symboly jsou stejné `[S, a]`, my ovšem potřebujeme, aby se provedla operace *shift*. Pomocí priorit lze sestrojit tyto 2 stromy (podle nastavení):

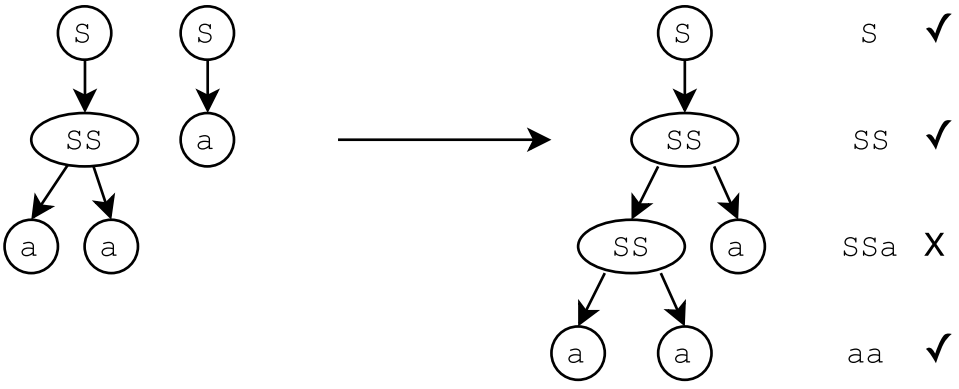


Z těchto stromů však ani jeden není ten správný. Pomocí priorit tento problém nevyřešíme.

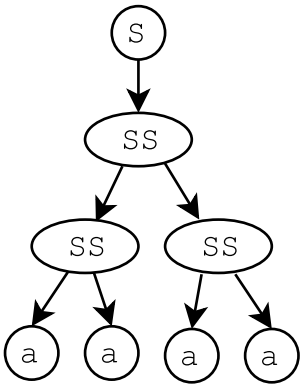
Zkusme podobně jako u LL syntaktické analýzy řešit konflikty v LR tabulce pomocí kontroly úrovní stromu. U LR SA to bude ověření, jestli lze dané stromy spojit (aplikovat pravidlo). Ukažme si to na předchozích konfliktech:



V tomto případě úrovně sedí, takže pravidlo použijeme. Další konflikt je následující:



Zde nám již kontrola úrovní neprojde, takže provedeme shift. Další konflikt už nenastane a výsledný strom vypadá takto:



Sestrojili jsme jediný správný strom, který vyhovuje kontrole úrovní. ■

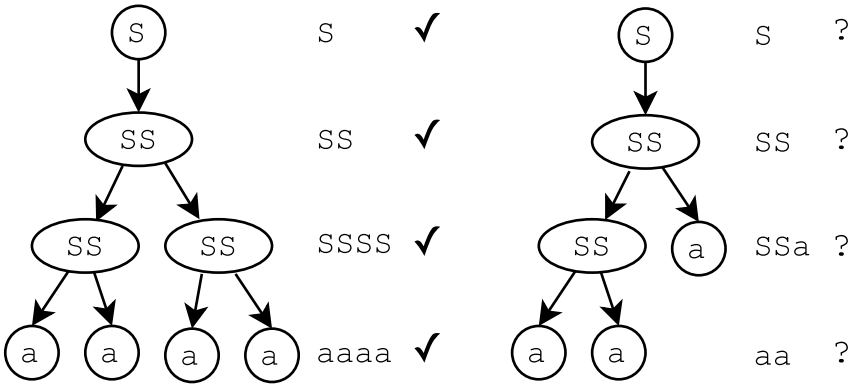
4.3.2 Nedeterministické kroky

Vraťme se ještě ke kroku, kdy se rozhodujeme mezi operacemi *shift* a *reduce* podle kontroly úrovní. V tu chvíli vlastně zkusíme provést *reduce* a pokud dojde ke konfliktu v úrovních, provedeme *shift*. Bohužel nejsme schopni určit, jestli operace *shift* povede ke konfliktu, proto v případě, že pokus o *reduce* projde, nezbývá než předpokládat, že *shift* je tou špatnou cestou. To ovšem neplatí obecně a pokud později narazíme na konflikt, znamená to pouze, že sestrojený strom je špatný, jiný strom by však mohl být správný. Nevíme tedy jistě, jestli řetězec do gramatiky opravdu nepatří.

Nechceme se ale ochudit o možnost pokusit se sestrojit správný strom - např. u výše uvedené gramatiky problém není. Musíme rozlišovat situace, kdy víme jistě, že řetězec patří nebo nepatří do gramatiky, a kdy to jistě nevíme.

4.3.3 Průběžná kontrola úrovní všech stromů

U výše uvedeného příkladu jsme ověřovali úrovně jen u nejlevějšího podstromu. Protože byl řetězec krátký, nepůsobilo to problémy. Když ovšem budeme kontrolovat delší řetězec, vzniknou nám stejné konflikty i v jiných podstromech. Je zde ale problém, že nevíme, jak budou později tyto podstromy připojeny k levému podstromu, a nejsme tudíž schopni kontrolovat od počátečního znaku.



Obrázek 4.1: Nerozpoznaný konflikt v pravém podstromě.

Abychom zajistili alespoň nějakou průběžnou kontrolu pravých podstromů, lze ověřit, jestli řetězec vůbec může být v daném kontrolním jazyce, tedy jestli je podřetězcem alespoň jednoho řetězce, který do kontrolního jazyka patří.

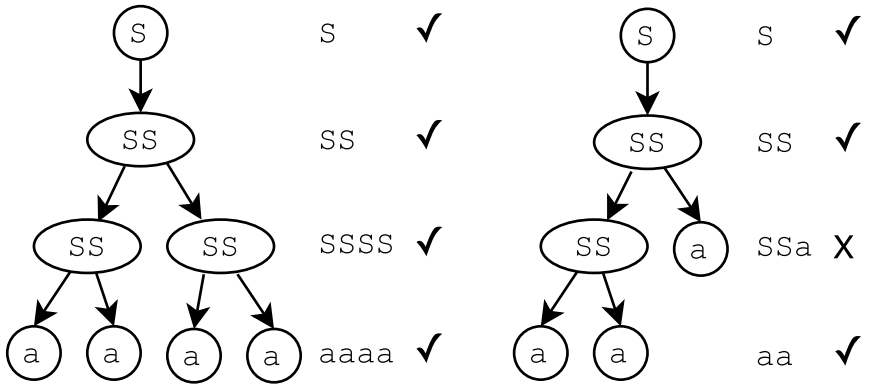
V případě modelu konečného automatu (který pro kontrolu úrovní budeme používat) lze tuto operaci provést tak, že místo jednoho stavu automatu budeme udržovat množinu stavů (ve kterých bychom se mohli nacházet). Takže vlastně zkusíme vyjít ze všech stavů automatu a ověříme, že alespoň z jednoho vede posloupnost přechodů odpovídající vstupu.

Algoritmus 4.1: Podřetězec pomocí konečného automatu.

Input: Konečný automat $M = (Q, \Sigma, R, s, F)$, vstupní řetězec a
Output: Úspěch nebo Chyba

```
1  begin
2      stavy := Q;
3      forall the znak c ∈ a do
4          forall the stav q ∈ stavy do
5              if přechod qc → q_new ∈ R then
6                  V množině Q nahraď q za q_new;
7              else
8                  Odeber q z Q;
9          if stavy = ∅ then
10             return Chyba;
11     return Úspěch;
```

Tento systém nám sice zajistí slabší kontrolu než u nejlevějšího podstromu, u výše uvedeného příkladu ale tento přístup zásadně pomůže:



Obrázek 4.2: Rozpoznaný konflikt v pravém podstromě.

4.3.4 Reduce-reduce konflikty

Podobným způsobem lze řešit také některé reduce-reduce konflikty. V těchto případech se dokonce můžeme chovat deterministicky, jelikož lze vyzkoušet všechna pravidla a pokračovat pouze v případě, že vyhovuje právě jedno. V jiných případech rovnou skončíme chybou.

4.3.5 Rychlost algoritmu

Jelikož při řešení konfliktů musíme kontrolovat úrovně derivačního stromu, zvyšuje se celková asymptotická složitost algoritmu. Při řešení konfliktů má na výkon vliv i velikost automatu kontrolního jazyka, jelikož u pravých podstromů musíme procházet velkou část těchto stavů.

Závěr

Průběžnou kontrolou úrovní samozřejmě nelze zpracovávat jakoukoliv gramatiku. Obecně problém nastává tehdy, když se o konfliktu ve stromě rozhoduje v době, kdy se ještě neprojeví. Takový konflikt je poté sice rozpoznán, ale nelze jej už vyřešit bez navracení.

LR syntaktická analýza již v základu dokáže zpracovávat větší množství bezkontextových gramatik. Ukázali jsme si, že kontrola úrovní je zde sice složitější, při správném přístupu však může přinést poměrně výrazné rozšíření vyjadřovací síly. Výsledná vyjadřovací síla je vyšší než u *deterministických bezkontextových jazyků* a nižší oproti *rekurzivně vyčíslitelným jazykům*. LR syntaktická analýza se ukazuje jako nejvhodnější pro zpracování gramatik řízených stromy a je využita v praktické části tohoto projektu.

Kapitola 5

Implementace

Výběr programovacího jazyka

Z čistě praktických důvodů byl pro implementaci vybrán jazyk Python (verze 3.4). Jde o jazyk skriptovací a vysokoúrovňový, lze tedy očekávat nižší rychlost oproti kompilovaným a nízkoúrovňovějším jazykům. Jelikož však tato aplikace slouží převážně pro demonstrační účely, není rychlost tím zásadním parametrem, a je tedy možné využít výhod, jež vysokoúrovňový jazyk nabízí.

Uživatelské rozhraní

Jelikož aplikace pracuje pouze s textovým vstupem, je program koncipován jako konzolová aplikace, ovládá se tedy výhradně přes příkazový řádek.

Vstupní parametry

Aplikace přijímá parametry ve standardním unixovém formátu, jejich přesný formát je uveden v souboru `README`.

Jediným povinným argumentem je vstupní gramatika, jež je načtena z externího souboru. Hlavní součástí tohoto souboru je řízená bezkontextová gramatika, volitelně poté můžeme přidat kontrolní jazyk ve formě konečného automatu nebo výčtu řetězců (ty jsou také převedeny na konečný automat). Konečný automat je v aplikaci determinizován. Volitelně zde lze definovat také priority operátorů. Formát tohoto souboru je detailně specifikován v souboru `README`.

Dále je nutné programu předat vstupní řetězec. Volitelným argumentem je možné aplikaci předat název externího souboru. Pokud není specifikován, je čten ze standardního vstupu. Tento vstupní řetězec je parsován konečným automatem, který přijímá terminální symboly gramatiky. Pokud lze vstupní symboly interpretovat pouze jedním způsobem, není nutné je oddělovat mezerami (ty jsou ignorovány).

Aplikace umožňuje detailně zvolit, jaké informace má tisknout při zpracování jazyka a zpracování řetězce. Ve výchozím stavu aplikace tiskne pouze chybová hlášení na standardní chybový výstup. Pomocí příslušného argumentu lze ale specifikovat tisk např. LR tabulky, použitých pravidel atp.

Chování aplikace

Pokud specifikujeme pouze řízenou gramatiku, aplikace se chová jako standardní LR syntaktický analyzátor a hlásí chyby při konfliktech LR tabulce. V případě, že přidáme priority operátorů, jsou povoleny *shift-reduce* konflikty, které jsou řešitelné pomocí priority operátorů. Pokud specifikujeme i kontrolní jazyk, aktivuje se průběžná kontrola úrovní a jsou povoleny *shift-reduce* i *reduce-reduce* konflikty. Tyto konflikty jsou poté řešeny za běhu tímto způsobem:

- V případě *shift-reduce* konfliktu se pokusíme vyloučit operaci *reduce*. Pokud je to možné, provedeme *shift*, jinak provedeme *reduce* a zapamatujeme si, že bylo jednáno nedeterministicky (viz sekce 4.3.2), což se projeví při pozdější chybě.
- Pokud jde o *reduce-reduce* konflikt, vyzkoušíme všechny možnosti a pokračujeme pouze pokud lze použít právě jedno pravidlo. Jinak končíme chybou.

Výstup aplikace

Primárním účelem aplikace je rozhodnutí, jestli řetězec patří do dané gramatiky, či nikoliv. Dále je však nutné ošetřit případy chybného vstupu, argumentů atp.

V případě, že vše proběhlo v pořádku a řetězec patří do gramatiky, je návratový kód `0`. Návratový kód odpovídá číslu chyby a je také vytištěn krátký popis chyby. Pokud se jedná o chybu ve vstupu, je připojena také informace o řádku a pozici v souboru. Podrobný popis návratových kódů je v souboru `README`.

Speciálním případem je chyba nedeterministického zpracování, která nastává v případě, že pro danou situaci neexistuje pravidlo, ale v minulosti bylo využito nedeterministického postupu, takže nejsme schopni určit, jestli řetězec do jazyka patří, či nikoliv. Tento problém je vysvětlen v sekci 4.3.2.

Vývoj a testování

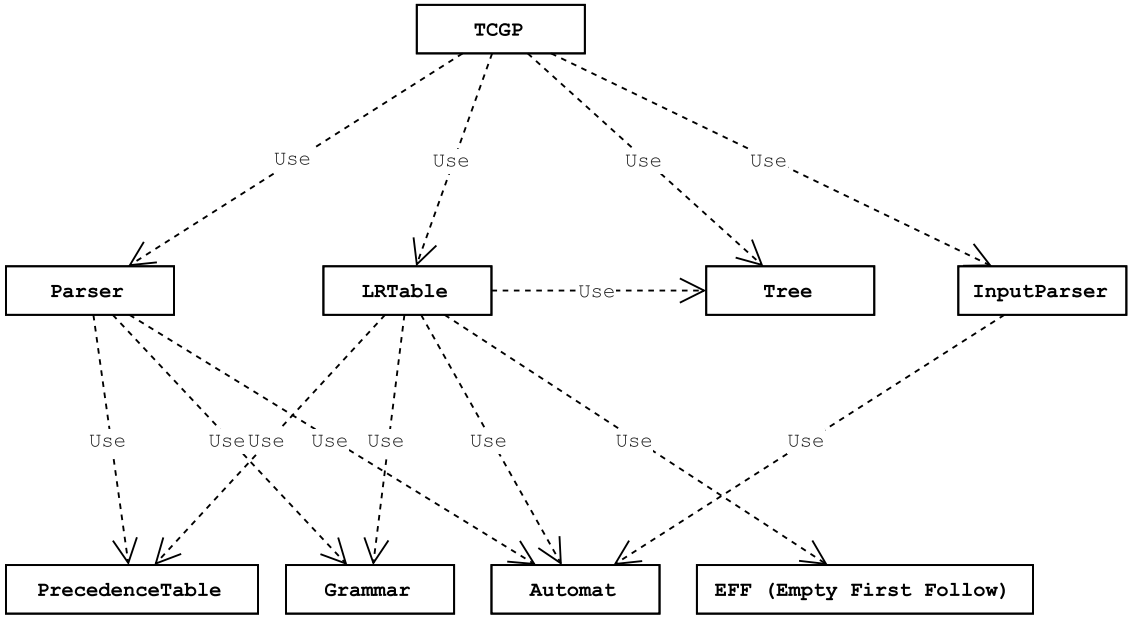
Vývoj probíhal v iteracích, kdy byla vždy navržena a implementována část projektu (např. jeden modul), přidány nové testy, zaměřující se na tuto část, a poté byla celá aplikace důkladně otestována.

Pro účely testování byla postupně navržena automaticky spustitelná sada testů typu black-box, kdy testujeme krajní a složité případy jednotlivých modulů i celkovou funkčnost.

Výkon

LR parser je v základní podobě schopen zpracovávat vstup v lineárním čase $O(n)$. Samotná kontrola úrovní zvýší asymptotickou složitost na $O(n \cdot log(n))$, při předpokladu, že nenarazíme na žádný konflikt (nejlepší případ). Při řešení konfliktů jsme však nuceni zkoušet spojování podstromů, u pravých podstromů ještě pracujeme se složitým procházením mnoha stavů automatu. Výsledná nejhorší složitost by se tedy blížila $O(n \cdot log(n) \cdot m)$, kde n je délka vstupního řetězce a m velikost konečného automatu.

Při návrhu a implementaci byl však kladen důraz hlavně na čitelnost a rozšiřitelnost kódu, a proto nebylo přistupováno k mnoha optimalizacím. Výsledný výkon bude o něco horší.



Obrázek 5.1: Zjednodušený diagram tříd.

Kapitola 6

Závěr

Tato práce se zaměřila na vývoj syntaktického analyzátoru, který zpracovává gramatiky řízené stromy. Přestože tyto gramatiky mají teoreticky vyjadřovací sílu *neomezených gramatik*, při praktickém nasazení jsme narazili na mnoho potíží, které vyjadřovací sílu snižují.

Při vývoji syntaktického analyzátoru jsme vycházeli ze stávajících metod a zkoumali jsme, jak bychom mohli tyto metody upravit pro kontrolu derivačního stromu. Tyto metody jsou podrobně rozebrány v sekci 3.3, kde jsme si ukázali, jaké rozdíly jsou v konstrukci derivačního stromu. Porovnáváme sílu těchto metod a je zde uvedeno, jaké problémy nastávají, když gramatiku nelze zpracovávat daným syntaktickým analyzátorem. U LR syntaktické analýzy prezentujeme také metodu priorit operátorů, která umožňuje řešit některé konflikty u nedeterministických gramatik.

Výsledky

V sekci 4.2 jsme upravili LL syntaktickou analýzu pro průběžnou kontrolu derivačního stromu a následně jsme se toho pokusili využít pro řešení některých konfliktů v LL tabulce. Lze konstatovat, že pomocí LL syntaktické analýzy můžeme pohodlně (a jednoduše) kontrolovat úrovně derivačního stromu již za běhu, jelikož je strom konstruován výhodným způsobem shora a zleva. Řešení konfliktů v LL tabulce však není příliš účinné, jelikož máme při aplikaci pravidla k dispozici velmi malou část stromu vytvořeného pravidlem, a proto nejsme ve většině případů schopni odhalit konflikt ihned. Tento poznatek společně s faktem, že LL syntaktická analýza již v základu nabízí nižší sílu než LR syntaktická analýza, vedl nakonec k opuštění LL syntaktické analýzy, jelikož jsme nebyli schopni její sílu výrazně zvýšit a i při využití průběžné kontroly derivačního stromu pro řešení konfliktů výrazně zaostávala za LR syntaktickou analýzou.

Sekce 4.3 se proto plně zaměřuje na LR syntaktickou analýzu. Ukázali jsme si, že zde je průběžná kontrola složitější, jelikož je strom konstruován zdola a máme v jednu chvíli více nepropojených stromů. Kontrolovat průběžně úrovně lze jednoduše pouze u nejlevějšího z nich. Jelikož však LR syntaktická analýza sestavuje derivační strom zdola a postupně vlastně dochází ke spojování jednotlivých podstromů, jsme při tomto spojování schopni zkontrolovat kompletní podstrom vytvořený daným pravidlem a je zde mnohem pravděpodobnější, že odhalíme případný konflikt ihned. Tato metoda přinesla úspěch při řešení některých konfliktů v LR tabulce, a to jak *shift-reduce*, tak i *reduce-reduce*.

V podsekci 4.3.3 je prezentována metoda, jak lze částečně kontrolovat i pravé podstromy.

Problém tkví v tom, že u těchto podstromů nemáme začátek řetězců úrovní, a proto nejsme schopni zahájit kontrolu úrovní z počátečního stavu konečného automatu. Chceme tedy alespoň ověřit, jestli je řetězec podřetězcem nějakého řetězce, který do kontrolního jazyka patří. Na začátku tedy nevíme, z jakého stavu vyjít, proto vyzkoušíme stavy všechny a vylučujeme ty, u kterých nejsme schopni pokračovat ve zpracovávání řetězce. Pokud máme na konci alespoň jeden stav, můžeme prohlásit, že řetězec je podřetězcem alespoň jednoho řetězce patřícího do kontrolního jazyka.

I když je tato metoda dosti těžkopádná, zajišťuje alespoň základní kotrolu pravých podstromů, což je pro průběžnou kontrolu zásadní.

Pomocí metod uvedených v této části lze zpracovávat i jazyky s nedeterministickou řízenou gramatikou, kde není možné konflikty řešit pomocí priority operátorů. Jako příklad takové gramatiky je uveden jazyk a^{2^n}, jenž jsme tímto přístupem schopni zpracovat.

Implementační část vychází z teoretických poznatků a představuje syntaktický analyzátor umožňující zpracovávání velké části *jazyků řízených stromy*. Podporuje zadávání kontrolního jazyka jak v podobě konečného automatu, tak i uživatelsky přivětivým jednoduchým výčtem vyhovujících řetězců. Umožněna je také definice priorit operátorů. Cílem je umožnit zpracovávání co největšího počtu gramatik přesahujících možnosti gramatik bezkontextových. Výslednou sílu lze ohraničit takto: je vyšší než u *bezkontextových deterministických gramatik* a nižší oproti *neomezeným gramatikám*.

Daní za velkou sílu syntaktického analyzátoru je zvýšení asymptotické složitosti zpracování řetězce (viz kapitola 5). Důvodem tohoto zpomalení je řešení konfliktů až za běhu, zde procházíme všechny podstromy, které jsou spojovány aplikovaným pravidlem, a jelikož nemusí jít o nejlevější podstrom, záleží také na velkosti konečného automatu. Tato oblast podle mého mínění nabízí prostor pro vylepšení.

Pro kontrolu úrovní derivačního stromu jsme využili poznatky uvedené v sekci 3.1, tedy odstranění ε-pravidel a determinizaci konečného automatu. Konečný automat je také využíván pro parsování tokenů a dokáže rozlišovat příchozí terminály i bez oddělovacích mezer.

Pomocí syntaktického analyzátoru se podařilo zpracovávat např. tyto výše demonstrované gramatiky, které nepatří do gramatik bezkontextových [4, str. 29]:

$$a^n b^n c^n$$

$$W\#W$$

$$a^{2^n}$$

Budoucí vývoj

Zde prezentovaný přístup pro zpracování *gramatik řízených stromy* by bylo možné na několika místech vylepšit. Například kontrola pravých stromů (sekce 4.3.3) je řešena dosti jednoduše a jistě by ji bylo možné zdokonalit. Vyloučení jakéhokoliv stavu již při začátku procházení by přineslo jak zrychlení programu, tak větší sílu, jelikož bychom odhalili více konfliktů. Pro tyto účely by bylo možné využít např. množinu follow, kde víme, jaké znaky se mohou nacházet za daným symbolem, a mohli bychom tedy vyloučit některé stavy pro určitý symbol ještě před zahájením zpracování řetězce. Možné by bylo také prozkoumat souvislost stavu zásobníkového automatu s úrovněmi - také v této oblasti bychom mohli dosáhnout zlepšení. Celkově větší provázání LR syntaktické analýzy a průběžné kontroly

úrovní by mohlo potenciálně přinést lepší výkon i zvýšení síly. Toto téma by však pravděpodobně vystačilo na samostatnou práci.

Pro zlepšení uživatelské přívětivosti by se také mohlo provést odstranění nedostupných a neukončujících stavů automatu. To by přineslo zvýšení účinnosti, ale pouze v případech, kdy automat kontrolující úrovně tyto stavy obsahuje.

Výsledná aplikace slouží hlavně pro demonstrační účely, ale použitá řešení by bylo možné využít např. pro specifické konstrukce programovacího jazyka, které nomohou být vyjádřeny pomocí bezkontextové gramatiky.

Literatura

[1] Aho, A. V.; a kol.: *Compilers: principles, techniques and tools. 2nd ed.* Boston: Addison Wesley, 2007, ISBN 0-321-48681-1.

[2] Hopcroft, J. E.; a kol.: *Introduction to Automata Theory, Languages, and Computation.* Boston: Pearson; Addison-Wesley, 2007, ISBN 0-321-47617-4.

[3] Knuth, D. E.: On the Translation of Languages from Left to Right [online]. http://www.cs.dartmouth.edu/%7Emckeeman/cs48/mxcom/doc/knuth65.pdf, 1965 [cit. 2016-3-29].

[4] Koutný, J.: *Gramatiky s omezenými derivačními stromy, phd thesis.* Brno, FIT VUT v Brně, 2002.

[5] Meduna, A.: *Formal languages and computation: models and their applications.* Taylor & Francis, New York, 2014, ISBN 978-1-4665-1345-7.

Přílohy

Příloha A

Obsah CD

Přiložené CD obsahuje:

- Zdrojové soubory syntaktického analyzátoru v jazyce python3
- Soubor README ve formátu .pdf i .md
- Testovací sadu
- Textovou zprávu ve formátu .pdf
- Zdrojové soubory textové zprávy ve formátu LaTeX